注: 1.阅读此篇前你需要先阅读我的之前一篇文章:《spring的aop原理源码解析》

测试代码:

```
@Repository
public class UserDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;
    public void insert(){
        String sql = "INSERT INTO `tbl_user`(username,age) VALUES(?,?)";
        String username = UUID.randomUUID().toString().substring(0, 5);
        jdbcTemplate.update(sql, username,19);

    }

}

@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    //事务注解 这个里面可以配置各种事务 例如:readOnly   rollbackFor等
    @Transactional
    public void insertUser(){
        userDao.insert();
        //otherDao.other();xxx
        System.out.println("插入完成...");
        int i = 10/0;
    }

}
//@EnableTransactionManagement 开启基于注解的事务管理功能
@EnableTransactionManagement
@ComponentScan("com.mgw.tx")
@Configuration
public class TxConfig {

    //数据源
    @Bean
    public DataSource dataSource() throws Exception{
        ComboPooledDataSource dataSource = new ComboPooledDataSource();
        dataSource.setUser("xxx");
        dataSource.setPassword("xxx");
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql://xxx:3306/xxx");
        return dataSource;
    }

    //jdbc模板类
```

```
    @Bean
    public JdbcTemplate jdbcTemplate() throws Exception{
        //Spring对@Configuration类会特殊处理；给容器中加组件的方法，多次调用都只是从容器中找组件
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource());
        return jdbcTemplate;
    }

    //注册事务管理器在容器中  事务管理器是必配的  否则容器无法得知你的事务管理策略
    @Bean
    public PlatformTransactionManager transactionManager() throws Exception{
        return new DataSourceTransactionManager(dataSource());
    }
}

//测试类
public class IOCTest_Tx {

    @Test
    public void test01(){
        AnnotationConfigApplicationContext applicationContext =
                new AnnotationConfigApplicationContext(TxConfig.class);

        UserService userService = applicationContext.getBean(UserService.class);

        userService.insertUser();
        applicationContext.close();
    }

}
```

分析@EnableTransactionManagement这个注解

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(TransactionManagementConfigurationSelector.class)
public @interface EnableTransactionManagement {

    boolean proxyTargetClass() default false;

    AdviceMode mode() default AdviceMode.PROXY;

    int order() default Ordered.LOWEST_PRECEDENCE;

}
```

其导入了这个TransactionManagementConfigurationSelector类

```
public class TransactionManagementConfigurationSelector extends
AdviceModeImportSelector<EnableTransactionManagement> {

    @Override
    protected String[] selectImports(AdviceMode adviceMode) {
        switch (adviceMode) {
            case PROXY:
                //导入AutoProxyRegistrar，ProxyTransactionManagementConfiguration组件
                return new String[] {AutoProxyRegistrar.class.getName(),
ProxyTransactionManagementConfiguration.class.getName()};
```

```
            case ASPECTJ:
                return new String[]
{TransactionManagementConfigUtils.TRANSACTION_ASPECT_CONFIGURATION_CLASS_NAME};
            default:
                return null;
        }
    }

}
```

总体来说@EnableTransactionManagement这个注解:

利用TransactionManagementConfigurationSelector给容器中导入组件

导入两个组件:AutoProxyRegistrar和ProxyTransactionManagementConfiguration

分析:

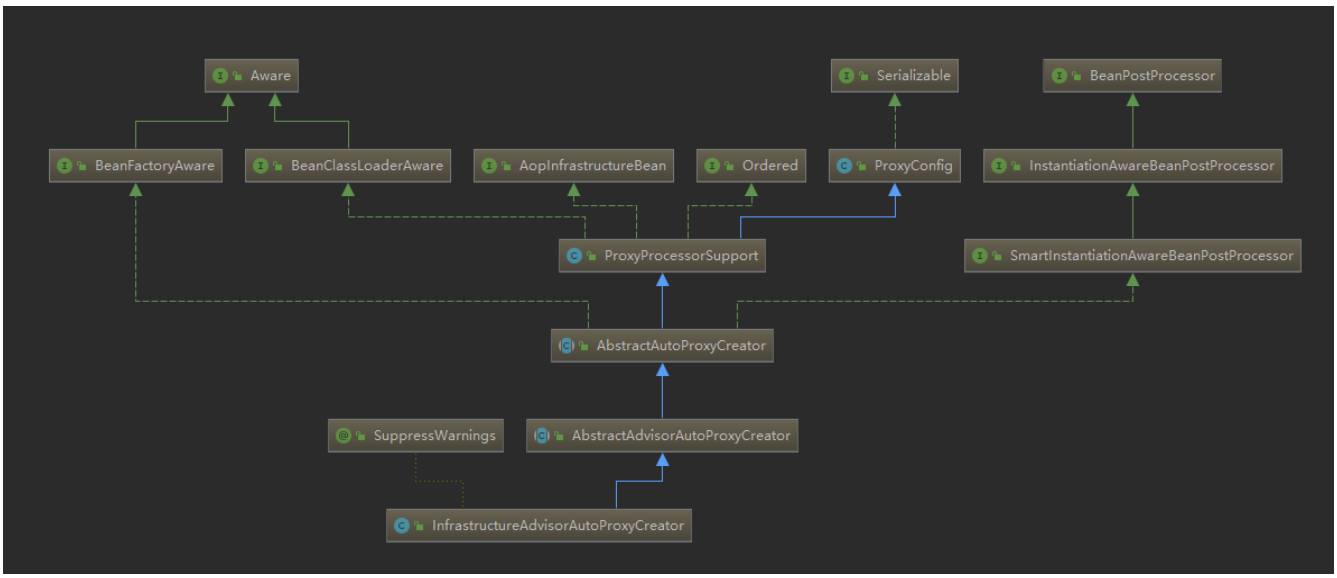
1.AutoProxyRegistrar组件

```
//注意:其继承了ImportBeanDefinitionRegistrar接口  其这个接口就是在工厂刚刚创建完毕后，工厂的后置处理器会回调这个
接口的方法为容器中注入bean信息
public class AutoProxyRegistrar implements ImportBeanDefinitionRegistrar {

    private final Log logger = LogFactory.getLog(getClass());


    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
BeanDefinitionRegistry registry) {
        boolean candidateFound = false;
        Set<String> annoTypes = importingClassMetadata.getAnnotationTypes();
        for (String annoType : annoTypes) {
            AnnotationAttributes candidate =
AnnotationConfigUtils.attributesFor(importingClassMetadata, annoType);
            if (candidate == null) {
                continue;
            }
            Object mode = candidate.get("mode");
            Object proxyTargetClass = candidate.get("proxyTargetClass");
            if (mode != null && proxyTargetClass != null && AdviceMode.class == mode.getClass() &&
                    Boolean.class == proxyTargetClass.getClass()) {
                candidateFound = true;
                if (mode == AdviceMode.PROXY) {
                    //调用这个自动代理创建器为容器中注册一个类
                    /*
                    registerOrEscalateApcAsRequired(InfrastructureAdvisorAutoProxyCreator.class,
registry, source)
                    发现了吗? 和aop创建时为其注入AnnotationAwareAspectJAutoProxyCreator这个类使用的方式一
模一样
                    而且其调用的工具类一摸一样  如果你细心点儿会发现aop创建类的方法就在下面
                    */
                    AopConfigUtils.registerAutoProxyCreatorIfNecessary(registry);
                    if ((Boolean) proxyTargetClass) {
                        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
                        return;
                    }
```

```
                }
            }
        }
        if (!candidateFound) {
            String name = getClass().getSimpleName();
            logger.warn(String.format("%s was imported but no annotations were found " +
                    "having both 'mode' and 'proxyTargetClass' attributes of type " +
                    "AdviceMode and boolean respectively. This means that auto proxy " +
                    "creator registration and configuration may not have occurred as " +
                    "intended, and components may not be proxied as expected. Check to " +
                    "ensure that %s has been @Import'ed on the same class where these " +
                    "annotations are declared; otherwise remove the import of %s " +
                    "altogether.", name, name, name));
        }
    }

}
```

给容器中注册一个 InfrastructureAdvisorAutoProxyCreator 组件

InfrastructureAdvisorAutoProxyCreator这个类是干嘛的?



注:InfrastructureAdvisorAutoProxyCreator继承关系图

利用后置处理器机制在对象创建以后，包装对象，返回一个代理对象（增强器），代理对象执行方法利用拦截器链进行调用

其实就和AnnotationAwareAspectJAutoProxyCreator这个功能类一模一样 没啥好说的了 过程都是一样的 可以去看aop那章的分析

2.ProxyTransactionManagementConfiguration组件

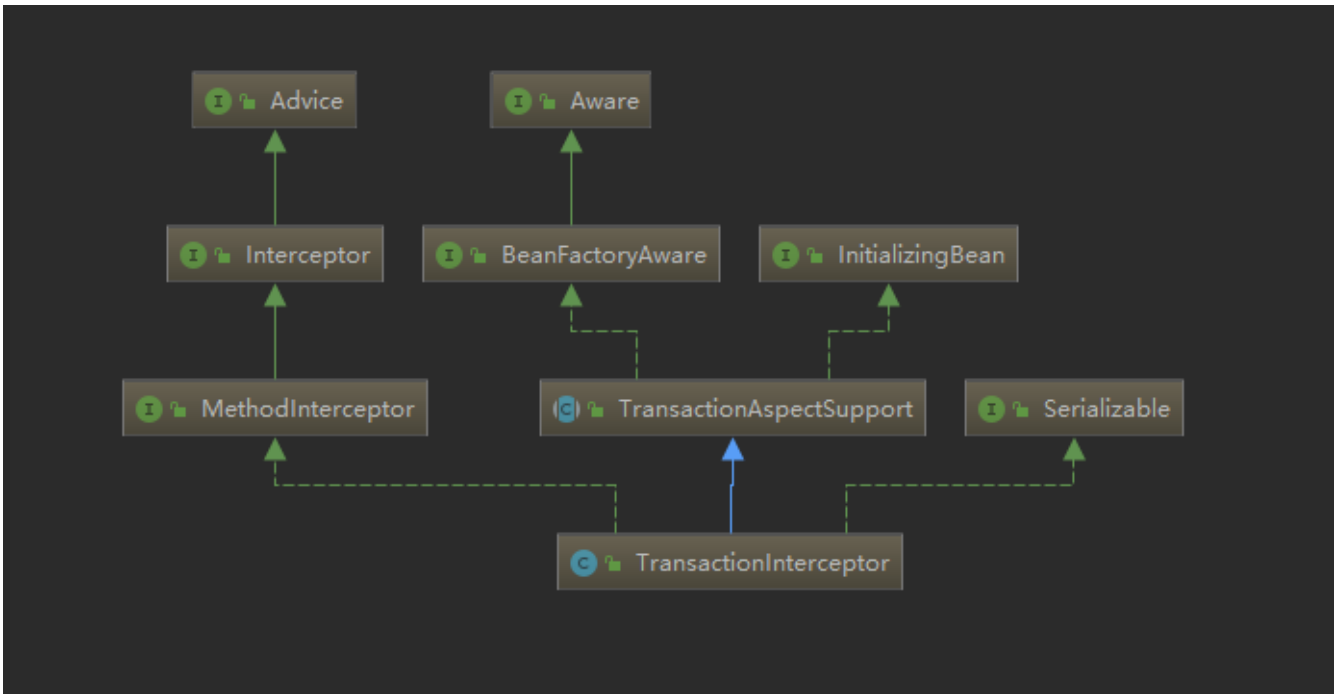
```
//注意这是一个配置类
@Configuration
public class ProxyTransactionManagementConfiguration extends
AbstractTransactionManagementConfiguration {

    //为容器加一个叫BeanFactoryTransactionAttributeSourceAdvisor的bean
    //这是一个事务增强器
```

```
    @Bean(name = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor() {
        BeanFactoryTransactionAttributeSourceAdvisor advisor = new
BeanFactoryTransactionAttributeSourceAdvisor();
        //添加事务属性信息
        advisor.setTransactionAttributeSource(transactionAttributeSource());
        //添加事务拦截器
        advisor.setAdvice(transactionInterceptor());
        advisor.setOrder(this.enableTx.<Integer>getNumber("order"));
        return advisor;
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    ////为容器加一个叫AnnotationTransactionAttributeSource的bean
    public TransactionAttributeSource transactionAttributeSource() {
        return new AnnotationTransactionAttributeSource();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    //为容器中增加一个TransactionInterceptor的bean
    //事务的拦截器
    public TransactionInterceptor transactionInterceptor() {
        TransactionInterceptor interceptor = new TransactionInterceptor();
        //保存事务的属性信息
        interceptor.setTransactionAttributeSource(transactionAttributeSource());
        if (this.txManager != null) {
            //保存事务管理器 还记得我们之前在自己的配置类中配置了一个事务管理器吗？就是那个
            interceptor.setTransactionManager(this.txManager);
        }
        return interceptor;
    }

}

public AnnotationTransactionAttributeSource(boolean publicMethodsOnly) {
    this.publicMethodsOnly = publicMethodsOnly;
    this.annotationParsers = new LinkedHashSet<TransactionAnnotationParser>(2);
    //注解的解析器 以SpringTransactionAnnotationParser这个解析器为例
    this.annotationParsers.add(new SpringTransactionAnnotationParser());
    if (jta12Present) {
        this.annotationParsers.add(new JtaTransactionAnnotationParser());
    }
    if (ejb3Present) {
        this.annotationParsers.add(new Ejb3TransactionAnnotationParser());
    }
}
//spring事务注解解析器
public class SpringTransactionAnnotationParser implements TransactionAnnotationParser,
Serializable {

    @Override
    public TransactionAttribute parseTransactionAnnotation(AnnotatedElement ae) {
        AnnotationAttributes attributes = AnnotatedElementUtils.getMergedAnnotationAttributes(ae,
Transactional.class);
```

```
        if (attributes != null) {
            return parseTransactionAnnotation(attributes);
        }
        else {
            return null;
        }
    }

    public TransactionAttribute parseTransactionAnnotation(Transactional ann) {
        return parseTransactionAnnotation(AnnotationUtils.getAnnotationAttributes(ann, false,
false));
    }
    //此方法就是解析各种注解性的信息  并封装进TransactionAttribute这个bean中
    protected TransactionAttribute parseTransactionAnnotation(AnnotationAttributes attributes) {
        RuleBasedTransactionAttribute rbta = new RuleBasedTransactionAttribute();
        Propagation propagation = attributes.getEnum("propagation");
        rbta.setPropagationBehavior(propagation.value());
        Isolation isolation = attributes.getEnum("isolation");
        rbta.setIsolationLevel(isolation.value());
        rbta.setTimeout(attributes.getNumber("timeout").intValue());
        rbta.setReadOnly(attributes.getBoolean("readOnly"));
        rbta.setQualifier(attributes.getString("value"));
        ArrayList<RollbackRuleAttribute> rollBackRules = new ArrayList<RollbackRuleAttribute>();
        Class<?>[] rbf = attributes.getClassArray("rollbackFor");
        for (Class<?> rbRule : rbf) {
            RollbackRuleAttribute rule = new RollbackRuleAttribute(rbRule);
            rollBackRules.add(rule);
        }
        String[] rbfc = attributes.getStringArray("rollbackForClassName");
        for (String rbRule : rbfc) {
            RollbackRuleAttribute rule = new RollbackRuleAttribute(rbRule);
            rollBackRules.add(rule);
        }
        Class<?>[] nrbf = attributes.getClassArray("noRollbackFor");
        for (Class<?> rbRule : nrbf) {
            NoRollbackRuleAttribute rule = new NoRollbackRuleAttribute(rbRule);
            rollBackRules.add(rule);
        }
        String[] nrbfc = attributes.getStringArray("noRollbackForClassName");
        for (String rbRule : nrbfc) {
            NoRollbackRuleAttribute rule = new NoRollbackRuleAttribute(rbRule);
            rollBackRules.add(rule);
        }
        rbta.getRollbackRules().addAll(rollBackRules);
        return rbta;
    }

    @Override
    public boolean equals(Object other) {
        return (this == other || other instanceof SpringTransactionAnnotationParser);
    }

    @Override
    public int hashCode() {
        return SpringTransactionAnnotationParser.class.hashCode();
    }
```

```
}
```

事务拦截器:他继承了MethodInterceptor这个方法拦截器 说明它最后就是一个方法拦截器 想起aop哪里了吗?



看下TransactionInterceptor这个类

```
public class TransactionInterceptor extends TransactionAspectSupport implements MethodInterceptor,
Serializable {

    /**
     * Create a new TransactionInterceptor.
     * <p>Transaction manager and transaction attributes still need to be set.
     * @see #setTransactionManager
     * @see #setTransactionAttributes(java.util.Properties)
     * @see #setTransactionAttributeSource(TransactionAttributeSource)
     */
    public TransactionInterceptor() {
    }

    /**
     * Create a new TransactionInterceptor.
     * @param ptm the default transaction manager to perform the actual transaction management
     * @param attributes the transaction attributes in properties format
     * @see #setTransactionManager
     * @see #setTransactionAttributes(java.util.Properties)
     */
    public TransactionInterceptor(PlatformTransactionManager ptm, Properties attributes) {
        setTransactionManager(ptm);
        setTransactionAttributes(attributes);
    }

    /**
     * Create a new TransactionInterceptor.
     * @param ptm the default transaction manager to perform the actual transaction management
     * @param tas the attribute source to be used to find transaction attributes
```

```
     * @see #setTransactionManager
     * @see #setTransactionAttributeSource(TransactionAttributeSource)
     */
    public TransactionInterceptor(PlatformTransactionManager ptm, TransactionAttributeSource tas)
{
        setTransactionManager(ptm);
        setTransactionAttributeSource(tas);
    }

    //执行代理方法的地方
    @Override
    public Object invoke(final MethodInvocation invocation) throws Throwable {
        // Work out the target class: may be {@code null}.
        // The TransactionAttributeSource should be passed the target class
        // as well as the method, which may be from an interface.
        Class<?> targetClass = (invocation.getThis() != null ?
AopUtils.getTargetClass(invocation.getThis()) : null);

        // Adapt to TransactionAspectSupport's invokeWithinTransaction...
        return invokeWithinTransaction(invocation.getMethod(), targetClass, new
InvocationCallback() {
            @Override
            public Object proceedWithInvocation() throws Throwable {
                return invocation.proceed();
            }
        });
    }


    //---------------------------------------------------------------------
    // Serialization support
    //---------------------------------------------------------------------

    private void writeObject(ObjectOutputStream oos) throws IOException {
        // Rely on default serialization, although this class itself doesn't carry state anyway...
        oos.defaultWriteObject();

        // Deserialize superclass fields.
        oos.writeObject(getTransactionManagerBeanName());
        oos.writeObject(getTransactionManager());
        oos.writeObject(getTransactionAttributeSource());
        oos.writeObject(getBeanFactory());
    }

    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        // Rely on default serialization, although this class itself doesn't carry state anyway...
        ois.defaultReadObject();

        // Serialize all relevant superclass fields.
        // Superclass can't implement Serializable because it also serves as base class
        // for AspectJ aspects (which are not allowed to implement Serializable)!
        setTransactionManagerBeanName((String) ois.readObject());
        setTransactionManager((PlatformTransactionManager) ois.readObject());
        setTransactionAttributeSource((TransactionAttributeSource) ois.readObject());
        setBeanFactory((BeanFactory) ois.readObject());
    }
```

```
}

protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final
InvocationCallback invocation)
            throws Throwable {

    // If the transaction attribute is null, the method is non-transactional.
    //先获取到事务的属性
    final TransactionAttribute txAttr =
getTransactionAttributeSource().getTransactionAttribute(method, targetClass);
    //获取到事务管理器
    final PlatformTransactionManager tm = determineTransactionManager(txAttr);
    //得到要执行的事务方法
    final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);

    if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
        // Standard transaction demarcation with getTransaction and commit/rollback calls.
        TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr,
joinpointIdentification);
        Object retVal = null;
        try {
            // This is an around advice: Invoke the next interceptor in the chain.
            // This will normally result in a target object being invoked.
            //执行目标方法
            retVal = invocation.proceedWithInvocation();
        }
        catch (Throwable ex) {
            // target invocation exception
            //如果发生异常 这里处理
            completeTransactionAfterThrowing(txInfo, ex);
            throw ex;
        }
        finally {
            cleanupTransactionInfo(txInfo);
        }
        //没有异常拿到返回值操作
        commitTransactionAfterReturning(txInfo);
        return retVal;
    }

    else {
        // It's a CallbackPreferringPlatformTransactionManager: pass a TransactionCallback in.
        try {
            Object result = ((CallbackPreferringPlatformTransactionManager) tm).execute(txAttr,
                    new TransactionCallback<Object>() {
                        @Override
                        public Object doInTransaction(TransactionStatus status) {
                            TransactionInfo txInfo = prepareTransactionInfo(tm, txAttr,
joinpointIdentification, status);
                            try {
                                return invocation.proceedWithInvocation();
                            }
                            catch (Throwable ex) {
                                if (txAttr.rollbackOn(ex)) {
                                    // A RuntimeException: will lead to a rollback.
                                    if (ex instanceof RuntimeException) {
                                        throw (RuntimeException) ex;
```

```
                                    }
                                    else {
                                        throw new ThrowableHolderException(ex);
                                    }
                                }
                                else {
                                    // A normal return value: will lead to a commit.
                                    return new ThrowableHolder(ex);
                                }
                            }
                            finally {
                                cleanupTransactionInfo(txInfo);
                            }
                        }
                    });

            // Check result: It might indicate a Throwable to rethrow.
            if (result instanceof ThrowableHolder) {
                throw ((ThrowableHolder) result).getThrowable();
            }
            else {
                return result;
            }
        }
        catch (ThrowableHolderException ex) {
            throw ex.getCause();
        }
    }
}
//获取事务管理器
protected PlatformTransactionManager determineTransactionManager(TransactionAttribute txAttr) {
    // Do not attempt to lookup tx manager if no tx attributes are set
    if (txAttr == null || this.beanFactory == null) {
        return getTransactionManager();
    }
    //如果事务属性可以获取到qualifier 这个属性是@Transactional注解中的一个属性用来指定事务管理器的 但是一般不配
置
    String qualifier = txAttr.getQualifier();
    if (StringUtils.hasText(qualifier)) {
        return determineQualifiedTransactionManager(qualifier);
    }
    else if (StringUtils.hasText(this.transactionManagerBeanName)) {
        return determineQualifiedTransactionManager(this.transactionManagerBeanName);
    }
    else {
        //获取默认的事务管理器 一般也是没有的
        PlatformTransactionManager defaultTransactionManager = getTransactionManager();
        if (defaultTransactionManager == null) {
            //从缓存中取 第一次肯定也是没的
            defaultTransactionManager =
this.transactionManagerCache.get(DEFAULT_TRANSACTION_MANAGER_KEY);
            if (defaultTransactionManager == null) {
                //从容易中获取 这个就是自己配置的那个事务管理器
                defaultTransactionManager =
this.beanFactory.getBean(PlatformTransactionManager.class);
                this.transactionManagerCache.putIfAbsent(
                        DEFAULT_TRANSACTION_MANAGER_KEY, defaultTransactionManager);
```

```
            }
        }
        return defaultTransactionManager;
    }
}
//处理异常的地方
protected void completeTransactionAfterThrowing(TransactionInfo txInfo, Throwable ex) {
    if (txInfo != null && txInfo.hasTransaction()) {
        if (logger.isTraceEnabled()) {
            logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() +
                    "] after exception: " + ex);
        }
        if (txInfo.transactionAttribute.rollbackOn(ex)) {
            try {
                //拿到事务管理器然后回滚
                txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
            }
            catch (TransactionSystemException ex2) {
                logger.error("Application exception overridden by rollback exception", ex);
                ex2.initApplicationException(ex);
                throw ex2;
            }
            catch (RuntimeException ex2) {
                logger.error("Application exception overridden by rollback exception", ex);
                throw ex2;
            }
            catch (Error err) {
                logger.error("Application exception overridden by rollback error", ex);
                throw err;
            }
        }
        else {
            // We don't roll back on this exception.
            // Will still roll back if TransactionStatus.isRollbackOnly() is true.
            try {
                txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
            }
            catch (TransactionSystemException ex2) {
                logger.error("Application exception overridden by commit exception", ex);
                ex2.initApplicationException(ex);
                throw ex2;
            }
            catch (RuntimeException ex2) {
                logger.error("Application exception overridden by commit exception", ex);
                throw ex2;
            }
            catch (Error err) {
                logger.error("Application exception overridden by commit error", ex);
                throw err;
            }
        }
    }
}
//没有异常时 拿到返回值后的处理
protected void commitTransactionAfterReturning(TransactionInfo txInfo) {
    if (txInfo != null && txInfo.hasTransaction()) {
        if (logger.isTraceEnabled()) {
```

```
            logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() +
"]");
        }
        //拿到事务管理器提交事务
        txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
    }
}
```

总结如下:

ProxyTransactionManagementConfiguration 所做事情的总结 1、给容器中注册事务增强器； 1）、事务增强器要用事务注解的信息，AnnotationTransactionAttributeSource解析事务注解 2）、事务拦截器： TransactionInterceptor；保存了事务属性信息，事务管理器； 他是一个 MethodInterceptor； 在目标方法执行的时候； 执行拦截器链； 事务拦截器： 1）、先获取事务相关的属性 2）、再获取PlatformTransactionManager，如果事先没有添加指定任何transactionmanger 最终会从容器中按照类型获取一个PlatformTransactionManager； 3）、执行目标方法 如果异常，获取到事务管理器，利用事务管理回滚操作 如果正常，利用事务管理器，提交事务